Exploring the Cognitive Foundations of Software Engineering

Yingxu Wang, University of Calgary, Canada
Shushma Patel, London South Bank University, UK

ABSTRACT

It is recognized that software is a unique abstract artifact that does not obey any known physical laws. For software engineering to become a matured engineering discipline like others, it must establish its own theoretical framework and laws, which are perceived to be mainly relied on cognitive informatics and denotational mathematics, supplementing to computing science, information science, and formal linguistics. This paper analyzes the basic properties of software and seeks the cognitive informatics foundations of software engineering. The nature of software is characterized by its informatics, behavioral, mathematical, and cognitive properties. The cognitive informatics foundations of software engineering are explored on the basis of the informatics laws of software and software engineering psychology. A set of fundamental cognitive constraints of software engineering, such as intangibility, complexity, indeterminacy, diversity, polymorphism, inexpressiveness, inexplicit embodiment, and unquantifiable quality measures, is identified. The conservative productivity of software is revealed based on the constraints of human cognitive capacity.

Keywords: Cognitive Models; Cognitive Informatics; Denotational Mathematics; Foundations; Informatics Laws; Nature of Software; Programming Psychology; Properties; Software Engineering; Software Science; Software Science

INTRODUCTION

Software engineering is an applied discipline of software science that adopts engineering approaches, such as established methodologies, processes, architectures, measurement, tools, standards, organisation methods, management methods, quality assurance systems and the like, in the development of large-scale software seeking to result in high productivity, low cost, controllable quality, and measurable development schedule (Bauer, 1972; Dijkstra, 1976; Brooks, 1987; McDermid, 1991; Perters and Pedrycz, 2000; Wang, 2007a; Wang and King, 2000). Software Science is a discipline that studies the theoretical framework of software as instructive and behavioral information, which can be embodied and executed by generic computers in...
order to create expected system behaviors and machine intelligence (Wang, 2007a, 2009a). The relationship between software science and software engineering can be described as that software science is theoretical software engineering; while software engineering is applied software science.

The object under study in software engineering and software science are software and program systems, which are a set of behavioral instructions for implementing a certain architectural layout of data objects and for embodying a set of expected behaviors on a universal computer platform for a required application. Large-scale software systems are highly complicated systems that have never been handled by mankind in engineering disciplines. It is recognized that software is a unique abstract artifact that does not obey any known physical laws (McDermid, 1991; Hartmanis. 1994; Wang, 2007a). For software engineering to become a matured engineering discipline like others, it must establish its own theoretical framework and laws, which are perceived to be mainly relied on cognitive informatics (Wang, 2002a, 2003a, 2007b) and denotational mathematics (Wang, 2008a), supplementing to computing science (Gersting, 1982; Lewis and Papadimitriou, 1998), information science (Shannon, 1948; Bell, 1953; Goldman, 1953; Wang, 2002a, 2003a), and formal linguistics (Chomsky, 1957, 1965; Wang, 2007a).

This paper explores basic properties of software and cognitive foundations of software engineering. The nature of software and software engineering is explored in the facets of the informatics, behavioral, and mathematical properties. The cognitive informatics foundations of software engineering are sought on the basis of a set of informatics laws of software. The fundamental cognitive constraints of software engineering on intangibility, complexity, indeterminacy, diversity, polymorphism, inexpressiveness, inexplicit embodiment, and unquantifiable quality measures are elaborated. Based on the basic research, a set of cognitive informatics principles for software engineering is established, such as the conservative productivity of software constrained by human cognitive capacity, the cognitive characteristics of software engineering, software engineering psychology, the cognitive mechanism of skill transformation in software engineering, the cognitive foundations of software quality theories, and the cognitive complexity of software.

BASIC PROPERTIES OF SOFTWARE AND SOFTWARE ENGINEERING

The nature of software has been perceived quite differently in research and practice of computing and software engineering. Although in the IT and software industries, software is perceived broadly as a concrete product, there are three types of metaphors in perceiving the nature of software, known as the informatics, mathematics, and intelligent behavior metaphors. With the product metaphor, a number of manufacturing technologies and quality assurance principles were introduced into software engineering. However, the phenomenon, which we are facing almost the same problems in software engineering as we dealt with 40 years ago, indicates a deficiency of the manufacture and mass production based metaphors on software and its development. Therefore, the nature of software and software engineering need to be systematically investigated.

The Informatics Properties of Software

Information is the third essence in modeling the natural world supplementing to matter and energy. According to cognitive informatics theory (Wang, 2002a, 2003a, 2007b), information is any property or attribute of entities in the natural world that can be abstracted, digitally represented, and mentally processed. Software is both behavioral information to designers and instructive information to computers. With the informatics metaphor, software may be perceived as follows.
Definition 1. Software is a kind of coded and instructive information that describes an algebraic process system of software behaviors and architectures in computing.

The above definition indicates a new way to explain the laws and properties that govern the behavior of software. In other words, the informatics metaphor provides a new approach to study the nature and basic properties of software in software engineering, which forms an important part of the cognitive informatics foundations of software engineering.

In conventional engineering disciplines, the common approach moves from abstract to concrete, and the final product is the physical realization of an abstract design. In software engineering, however, the approach is reversed. The final software product is the virtualization and abstraction, by binary streams, of a set of original real-world requirements. The only tangible part of a software implementation is its storage media or its run-time behaviors. This is probably the most unique feature of software engineering.

There are twelve fundamental informatics properties identified in cognitive informatics, as described in Table 1, which constrain the behaviors of software and its quality (Wang, 2006). Since software is an abstract artifact that can be modeled and characterized only by information, all the informatics properties are applicable to software and its developing processes in software engineering.

The Intelligent Behavioral Properties of Software

A software system, to some extent, can be perceived as a virtual agent of human brains, because it is created to do something repeatable, to extend human capability, reachability, or memory. Conventional machines are invented to extend human physical capability, while modern information processing machines, such as computers, communication networks, and robots, are developed for extending human intelligence, memory, and the capacity for information processing. Therefore, any machine that may implement a part of human behaviors and actions in information processing is significantly important.

It is recognized (Wang, 2007b, 2009b) that the basic approaches to implement intelligent behaviors can be classified as shown in Table 2, where software for computation is the third approach to simulate and implement the natural intelligence by programmed logic. This indicates that software is a partial implementation of the natural intelligence and human behaviors, and a subset of simulated human intelligent behaviors described by programmed instructive information. Therefore, software is the simulation and execution of human behaviors, and the extension of human capability, reachability, persistency, memory, and information processing speed.

For further explain the nature of software, the following three situations where a software system is needed may be considered.

Theorem 1. The needs for software determined by the following three conditions are necessary and sufficient:

a. The repeatability: Software is required when one needs to do something for more than once.
b. The flexibility or programmability: Software is required when one needs to repeatedly do something not exactly the same.
c. The run-time determinability: Software is required when one needs to flexibly do something by a series of choices on the basis of varying sequences of events determinable only at run-time.

Theorem 1 indicates that the above three situations, namely repeatability, flexibility, and run-time determinability, form the necessary and sufficient conditions that warrant the requirement for a software solution in computing. Although repeatability is one of the most premier needs for a software solution, it is not the only sufficient condition for requiring software, because repeatability may also
Table 1. The cognitive informatics properties of software

<table>
<thead>
<tr>
<th>No.</th>
<th>Informatics properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Abstract artifacts</td>
<td>Information is an abstract artifact that can be elicited from the physical entities of the natural world. New information may be derived based on existing information.</td>
</tr>
<tr>
<td>2</td>
<td>Cumulativeness</td>
<td>Information is not conservative but cumulative, because information may be generated, reproduced, destroyed, and cumulated. The cumulativeness of information is the most significant attribute of information that human beings rely on for evolution.</td>
</tr>
<tr>
<td>3</td>
<td>Lossless reusable</td>
<td>Information is the unique artifact that can be reused by multiple users at the same time without loss in quantity and degradation in quality.</td>
</tr>
<tr>
<td>4</td>
<td>Dimensionless</td>
<td>Information has no physical size. No matter how large or small of the physical entities, their abstract representations or the cognitive visual objects occupy a similar sight frame; only the resolutions may be different.</td>
</tr>
<tr>
<td>5</td>
<td>Weightless:</td>
<td>A direct corollary based on Property 4 is that the weight of information is always zero.</td>
</tr>
<tr>
<td>6</td>
<td>Transformability between I-M-E</td>
<td>According to the I-M-E model, the three basic essences of the world are predicated to be transformable between each other. Any discovery about the unknown transformability will result in a significant evolution in software engineering and information science.</td>
</tr>
<tr>
<td>7</td>
<td>Multiple representation forms</td>
<td>Information can be represented in multiple forms by different means of abstraction.</td>
</tr>
<tr>
<td>8</td>
<td>Multiple carrying media</td>
<td>Parallel to Property 7, information can be carried by various media. It is noteworthy that a certain media may carry one or more forms of information. Correspondingly, a given form of information may be carried by different media.</td>
</tr>
<tr>
<td>9</td>
<td>Multiple transmission forms</td>
<td>The possible transmission forms of information are passing, broadcasting, gathering, and networking. The fast development of the Internet indicates that networking is the most advanced form of communications.</td>
</tr>
<tr>
<td>10</td>
<td>Generality of sources</td>
<td>The sources of information are widely generic. Information is formed by the combination between physical entities, abstract objects, and relations between them. According to the Object-Attribute-Relation (OAR) Model (Wang, 2007c), most new information can be elicited from relations among objects.</td>
</tr>
<tr>
<td>11</td>
<td>Conservation of information entropy and thermal entropy</td>
<td>In any system, the sum of the information entropy and the thermal entropy is a constant.</td>
</tr>
<tr>
<td>12</td>
<td>Unique quality attributes</td>
<td>To model the quality of software and information, a set of informatics-based quality attributes such as completeness, correctness, consistency, proper representation, clearness, feasibility, and verifiability have been identified. From this new angle, software quality can be defined as the achievement of the above inherent attributes for software architecture, static and dynamic behaviors.</td>
</tr>
</tbody>
</table>

Table 2. Approaches to Implement Intelligent Behaviors

<table>
<thead>
<tr>
<th>No.</th>
<th>Means</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Biological organisms</td>
<td>Naturally grown</td>
</tr>
<tr>
<td>2</td>
<td>Silicon automata</td>
<td>Wired</td>
</tr>
<tr>
<td>3</td>
<td>Information processors</td>
<td>Programmed</td>
</tr>
<tr>
<td>4</td>
<td>Other means</td>
<td>Hybrid</td>
</tr>
</tbody>
</table>
be implemented by wired logic or hardware. Therefore, the flexibility and run-time determinability, particularly the latter, are necessary and sufficient for the usage of software. The third situation may be also considered as the non-determinism at compile-time or design-time. This feature is the fundamental issue in computation that determines the complexity of programming.

The Mathematical Properties of Software

The mathematical metaphor of software is widely adopted in computational science in which software is perceived as a mathematical entity or a stored programmed logic on computing hardware (von Neumann, 1946; Dijkstra, 1976; Lewis and Papadimitriou, 1998; Hartmanis, 1994; Hoare, 1969, 1986; Hoare et al., 1987; Wang, 2007a, 2008f, 2008k). A powerful concept toward the understanding of the nature of software, introduced by C.A.R. Hoare, is the term process (Hoare, 1978), by which the behaviors of a software system can be perceived as a set of processes composed with given rules defined in an algebraic form. It is found that a process can be formally modeled by a set of embedded relational statements (Wang, 2006, 2007a). Based on them, a program can be formally modeled by a set of embedded relational processes.

**Definition 2.** A process $P$ is an embedded relational composition of a list of $n$ meta-statements $p_i$ and $p_j$, $1 \leq i < n$, $1 < j \leq m = n+1$, according to certain composing relations $r_{ij}$, i.e.:

$$
P = \bigcap_{i=1}^{n-1} R(p_i, r_{ij}, p_j), j = i+1
$$

$$
= (\ldots (((p_1, r_{12}, p_2) r_{23}, p_3) \ldots r_{n-1,n}, p_n))
$$

(1)

where the big-R notation (Wang, 2008b) is adopted that describes the nature of processes as the building blocks of programs.

With the formal process model as defined above, a generic mathematical model of programs can be derived below.

**Definition 3.** A program $\varphi$ is a composition of a finite set of $m$ processes according to the time-, event-, and interrupt-based process dispatching rules of RTPA, i.e.:

$$
\varphi = \bigcap_{k=1}^{m} (@e_k \mapsto P_k)
$$

(2)

Definitions 2 and 3 indicate that a program is naturally an embedded relational algebraic entity, where a statement $s$ in a program is an instantiation of a meta-instruction of a programming language that executes a basic unit of coherent function and leads to a predictable behavior.

**Theorem 2.** The Embedded Relational Model (ERM) states that a software system or a program $\varphi$ is a set of complex embedded relational processes, in which all previous processes of a given process form the context of the current process, i.e.:

$$
\varphi = \bigcap_{k=1}^{m} (@e_k \mapsto P_k)
$$

$$
= \bigcap_{k=1}^{m} (@e_k \mapsto \bigcap_{j=1}^{n-1} R(p_{(k)}, r_{j(k)}, p_{j(k)})), j = i+1
$$

(3)

**Proof.** Theorem 2 can be directly proven on the basis of Definitions 2 and 3. Substituting $P_k$ in Definition 3 with Eq. 1, a generic program $\varphi$ obtains the form as a series of embedded relational processes as presented in Theorem 2.

The ERM model presented in Theorem 2 reveals that a program is a finite and nonempty set of embedded binary relations between a current statement and all previous ones that formed the semantic context or environment of computing. Theorem 2 provides a unified mathematical model of software, which is a formalization of the well accepted but infor-
mal process metaphor for software systems in computing.

According to Theorem 2, a program can be reduced to the composition of a finite set of \( k \) processes at the component level. Then, each of the processes can be further reduced to the composition of a finite set of \( n \) statements at the bottom level. The definitions, syntaxes, and formal semantics of each of the meta-processes and process relations may be referred to RTPA. A complex process and a program can be derived from the meta-processes by the set of algebraic process relations.

A general taxonomy of the usages of computational mathematics can be derived on the basis of their relations with natural languages. It is recognized that languages are the basic means of thinking (Chomsky, 1957, 1965). Although they can be rich, complex, and powerfully descriptive, natural languages share the common and basic mechanisms known as \textit{to be}, \textit{to have}, and \textit{to do} (Wang, 2007a, 2008a). All mathematical means and forms, in general, are an abstract description of these three categories of human or system behaviors and their common rules. Taking this view, mathematical logic may be perceived as the abstract means for describing \textit{‘to be’} expressions, set theory for describing \textit{‘to have,’} and functions for describing \textit{‘to do.’} This is a fundamental view toward the formal description and modeling of human and system behaviors in general, and software behaviors in particular.

Table 3 summarizes the usages of classic and denotational mathematics, which presents a fundamental view toward the modeling and expression of natural and machine intelligence in general, and software system in particular. Table 3 also indicates that only the logic- and set-based approaches are inadequate to deal with the entire problems in complex software and intelligent systems.

**FUNDAMENTAL COGNITIVE CONSTRAINTS OF SOFTWARE ENGINEERING**

Software engineering is a unique and probably the most complicated engineering discipline with fundamental cognitive, organizational, and resources constraints. These constraints are inherent due to its intangibility, intricate inner connections, the cognitive difficulty of software and their dependency on systems, diversity, and human. The study on the fundamental constraints of software engineering is helpful to: a) Understand the fundamental problems in software engineering, b) Guide the development of software engineering theories and methodologies, and c) Evaluate newly proposed software engineering theories, principles, and techniques.

**Definition 4.** The cognitive constraints of software engineering are a set of innate cognitive attributes of software and the nature of the problems in software engineering that create the intricate relations of software objects and make software engineering inherently difficult.

**Table 3. Basic expressive power and mathematical means in system modeling**

<table>
<thead>
<tr>
<th>Basic expressive power in system modeling</th>
<th>Mathematical means</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Classic mathematics</td>
<td>Denotational mathematics</td>
</tr>
<tr>
<td>To be</td>
<td>Logic</td>
<td>Concept algebra</td>
</tr>
<tr>
<td>To have</td>
<td>Set theory</td>
<td>System algebra</td>
</tr>
<tr>
<td>To do</td>
<td>Functions</td>
<td>RTPA</td>
</tr>
</tbody>
</table>

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
The following subsections identify and elaborate eight cognitive constraints of software engineering, such as intangibility, complexity, indeterminacy, diversity, polymorphism, inexpressiveness, inexplicit embodiment, and unquantifiable quality measures.

a. Intangibility

**Definition 5.** Intangibility is a basic cognitive constraint of software engineering that states software is an abstract artifact, which is not constituted by physical objects or presence, and is difficult to be defined or expressed.

The intangibility of software refers to all aspects of software and its development. That is, none of the entire software engineering processes, such as problem representation, requirements description, design, and implementation, is tangible.

b. Complexity

**Definition 6.** Complexity is a basic cognitive constraint of software engineering that states software is innately complex and its intricate internal connections and external couplings make it extremely difficult to be expressed or cognized.

The complexity of software refers to the complexities of its architectures, behaviors, and environments. The *architectural complexity* is the innate complexity of a software system with its data objects and their external/internal representations. The *behavioral complexity* is the complexity of a software system with its processes and their inputs/outputs. The *environmental complexity* is the complexity of a software system with its platform, related interacting processes, and users.

The most unique feature of software complexity is its intricate interconnection among components, functions, operations, and data objects. A small change in one place may result in multiple and unpredictable consequences in other places. This type of problem propagation due to intricate interconnection and coupling is a major challenge for system architects, programmers, and managers in software engineering.

The integration of a large-scale software system may easily result in a situation where no single person in the team may understand the system. During this phase, the project leader and system architect may lack sufficient detail knowledge about the implementation of the system, while individual programmers may lack the knowledge of a holistic view that treats the system as a whole with interfaces to other subsystems and components. This is a great challenge and a critical phase of human comprehension capability that often results in major failures after almost all resources have been exhausted in large-scale software projects.

c. Indeterminacy

**Definition 7.** Indeterminacy is a basic cognitive constraint of software engineering that states the events, behaviors, or their sequence of occurrence in a software system are not fully determinable on the basis of a given algorithm during design time. Instead, some of them may only be determinable until run-time.

The indeterminacy constraint indicates that, in general, a large portion of software behaviors and/or their sequence of occurrence are unpredictable at design time or compile time. Although the behavior space and all possible events are predictable, the order of events and the behaviors triggered by the chain of events will be greatly varying at run-time. Therefore, indeterminacy makes software design, implementation, and testing extremely difficult, because it results in a huge behavior space for a given software system, and its complete verification via testing are impossible sometimes.

Dijkstra discussed the special case of indeterminacy in automata where a given event to a finite state machine in a context may trigger no action because the machine cannot decide explicitly which action should be executed based on the given input and current state of the machine (Dijkstra, 1968b/1975). This kind
of phenomena occurs in operating system, agent systems, compiler design, and real-time systems, where additional information for decision making or an arbitrary decision needs to be adopted in the machine.

d. Diversity

**Definition 8.** Diversity is a basic cognitive constraint of software engineering that states the great variety of software in terms of types, styles, architectures, behaviors, platforms, application domains, implementation techniques, usability, reliability, and quality.

The extremely wide application domain of software dominates the cognitive complexity of software engineering and the knowledge requirement for architects and programmers who design and implement software systems. The diversity of software also refers to its types, which can be classified into system software, tools (compilers, code generators, communication/networking software, database management systems, and test software), and application software. The latter can be further categorized into transaction processing software, distributed software, real-time software, databases, and web-based software. It is noteworthy that there are fundamental differences between the technical, systematic, economic, organizational properties of system software and application software (Wang, 2007a).

e. Polymorphism

**Definition 9.** Polymorphism is a basic cognitive constraint of software engineering that states the approaches and styles of both software design and implementation are multifaceted and polyglottic.

Definition 9 indicates that the potential solution space of software engineering can be very large because both design and implementation have a great many options. According to the problem solving theory in cognitive informatics, software design and development is an open-end problem, which is similar to a creation process, where both possible solutions and paths that lead to one of the solutions are unknown and highly optional (Wang, 2008c; Wang and Chiew, 2008).

As that of the polysolvability for design, the polymorphism of software implementation refers to the cognitive phenomenon that approaches to implement a given design are not necessarily single. Many factors influence the solution space such as programming languages, target machine languages, coding styles, data models, memory allocations, and executing platforms. Any change among these factors may result in a different implementation of a software system.

**Theorem 3.** Polymorphous solutions state that the solution space SS of software engineering for a given problem is a product of the number of all design options D and the available implementation options I, i.e.:

\[ SS = D \cdot I \]

(4)

Therefore, the size of \( SS \), \( N_{SS} \), is determined by the numbers of both \( D \) and \( I \), i.e.:

\[ N_{SS} = N_D \cdot N_I \]  

(5)

On the basis of Theorem 3, a basic software engineering principle can be derived as follows.

**Corollary 1.** It is hard to prove technically and/or economically a certain software system is the optimal solution rather than a sound one constrained by the huge size of the solution space, which is known as the no number one principle in software engineering.

The polymorphic characteristic of the solution space of software engineering contributes greatly to the complexity of both theories and practices of software engineering.
f. **Inexpressiveness**

Software system requirements and specifications need to be essentially expressed in three aspects known as the *architecture*, *static behaviors*, and *dynamic behaviors* of the system.

**Definition 10.** Inexpressiveness is a basic cognitive constraint of software engineering that states software architectures and behaviors are inherently difficult to be expressed, modeled, represented, and quantified formally and rigorously.

Because software represents a set of instructive behavioral information, unless the behaviors and the underpinning architecture can be expressed rigorously and explicitly, no developer and machine may understand the requirement correctly and completely. Therefore, a new form of denotational mathematics (Wang, 2007a) is needed for system specification and refinement, of which typical paradigms are concept algebra (Wang, 2008g), system algebra (Wang, 2008h), and Real-Time Process Algebra (RTPA) (Wang, 2008i, 2008j).

g. **Inexplicit Embodiment**

Because software is intangible, the only way to make it embodied is to adopt more expressive means such as formal notations, programming languages, and visual diagrams.

**Definition 11.** Inexplicit embodiment is a basic cognitive constraint of software engineering that states architectures and behaviors of software systems should be explicitly described by coherent symbolic notations in order to be processed and executed by computers.

According to the Hierarchical Abstraction Model (HAM) of software engineering (Wang, 2008d), any notation or diagram that cannot explicitly describe the architecture and behaviors of software systems, or that highly depends on human interpretation of implied instructions, is inadequate. According to the explicit criterion, diagram-based techniques may be useful for describing conceptual models of software systems particularly for nonprofessionals, but it is unlikely to be an expressive and rigorous basis for future automatic code generation systems, because too much design and implementation information are implied rather than explicitly expressed. Although, machines are capable of carrying out translations between explicit specifications and code in order to improve software productivity, no machine may help to extend inadequate system specifications or to comprehend inexplicit system designs implied in the software architectural and behavioral information.

h. **Unquantifiable Quality Measures**

Quality is a generic measure of the degree of excellence of a product or service against a given standard. More specifically, quality is a common attribute of any product or service that characterizes the quantity of both utility and durability of the product or service. Determined by the complexity, diversity, and polymorphism constraints discussed earlier, the quality of software is a multifaceted entity and some facets of it are application specific.

**Definition 12.** Unquantifiable quality measures are a basic cognitive constraint of software engineering that states the model of software quality has intricate facets and is difficult to be quantitatively modeled and measured.

Software quality can be perceived from a relative point of view as the conformity of a software system to its specifications (design models). Therefore, software quality is inversely proportional to the differences between the behaviours and performance of a software system and those required in the specifications. However, many quality attributes of software, such as design quality, usability, implementation efficiency, and reliability, cannot be quantified, thus immeasurable.

A basic quality principle is “no measurement, no quality control.” The factor that it is
impossible to measure all quality attributes of a large-scale software system indicates that we are not completely in control of the development of such systems. Some qualitative or informal validation and evaluation techniques, such as review and prototyping (Boehm et al., 1984; McDermid, 1991; Arnowitz et al., 2006), are adopted in software engineering. Practitioners and users seem to be used to this situation. Therefore, measurement theories and methodologies for software systems had never been a central focus in software engineering, particularly because of its inherent difficulty in this area (Wang, 2003c).

**Definition 13.** Quality $Q$ is a generic and collective attribute of a product, a service, or a system that is proportional to both its average utility $U$ and the available duration $T$ of the utility, i.e.:

$$Q = U \cdot T \quad [\text{Fh}]$$

where the unit of utility is function (F), and the unit of duration is hour (h), and these result in the unit of quality as Function-hour, shortly Fh.

According to Definition 13, for a given product, service, or system, there is no quality if there is a lack of either utility ($U = 0$) or availability of the utility ($T = 0$). The quality given in Definition 13 is the average quality with a static view. A more generic form of quality for representing the dynamic aspect of quality as a function of time is given below.

**Definition 14.** A generic dynamic utility function $U(t)$ is an inverse exponential function over time, i.e.:

$$U(t) = U(1-e^{-T})[F]$$

where both $U$ and $T$ are a positive constant.

With the above definition of dynamic utility $U(t)$, the value of the dynamic quality can be determined by the following principle.

**Theorem 4.** The integrated quality with a dynamic utility, $Q(t)$, is an integral of the utility function $U(t)$ over the entire lifecycle of the utility $[0, T]$, i.e.:

$$Q(t) = \int_0^T U(t)dt$$

$$= \int_0^T U(1-e^{-T})dt$$

$$= U(e^{-T} + T - 1)$$

$$= UT - U(1-e^{-T})$$

$$= Q - U(1-e^{-T}) \quad [\text{Fh}]$$

(8)

where $U$ is the initial quality of a product, service, or system.

Theorem 4 shows that the integrated quality of a dissimilating utility system or product is always smaller than that of constant utility.

**COGNITIVE INFORMATICS PRINCIPLES FOR SOFTWARE ENGINEERING**

The cognitive constraints of software engineering as discussed in Section 3 identified the primary constraints of software engineering. On perceiving the cognitive properties of software as instructive intelligent behaviors, this section elaborates the cognitive informatics principles for software engineering, encompassing those of the conservative productivity, cognitive characteristics, software engineering psychology, cognitive skill transformation, quality assurance, and cognitive complexity of software.

**The Conservative Productivity in Software Engineering**

A profound discovery in software engineering is that the productivity of software development is conservative due to the cognitive mechanism in which abstract artifacts need to be represented physiologically in the brain via
naturally grown neural synaptic connections in the brain (Wang and Wang, 2006). In other words, software development productivity is constrained by natural laws rather than by human subjective intentions. The fact that before any program is composed, an internal abstract model must be created inside the brain (Wang, 2007c) reveals the most fundamental constraint of software engineering, i.e., software is generated and represented in the brain before it can be transferred into the computer. Because the growth rate of the human neural system is naturally constrained, as described by the 24-hour law (Wang and Wang, 2006), it is very hard to dramatically improve the productivity of software development.

The above theory is supported by empirical data and experience. According to the statistics of software engineering literature (Albrecht, 1979; Boehm, 1987; Jones, 1981, 1986; Livermore, 2005), the average productivity of software development was about 1,300 LOC/person-year in the 1970s, 2,500 LOC/person-year in the 1980s, and 3,000 LOC/person-year in the 1990s, where management, quality assurance, and supporting activities are included, and LOC is the unit of the symbolic size of software in terms of Line of Code. It is obvious that the productivity in software engineering has not been increased remarkably in the last four decades independent of the advances in programming languages and tools. In other words, no matter what kind of programming language is used, as long as they are for human programming, there is no difference in principle. This assertion can be proved by asking the following question: Have you ever known an author in literature who is productive because he/she writes in a particular natural language?

Productivity of software development is the key to cope with all the cognitive, time, and resources constraints in software engineering, because the other constraints may be overcome as a result of the improvement of software engineering productivity.

**Lemma 1.** The key approach to improve software development productivity are:

a. To explicitly express software architectures and behaviors in *denotational mathematics*;
b. To investigate the theories of rational software engineering organization; and
c. To develop tools that enable *automatic software code generation* based on the denotational system models.

The most significant and unique characteristic of software engineering lays on the need for the contemporary denotational mathematics in order to rigorously and explicitly model the architectures and behaviors of software systems and to reduce the cognitive complexity of software engineering, which are challenging the limitation of human cognitive capacity in large-scale software development.

**The Cognitive Characteristics of Software Engineering**

The following cognitive characteristics dominate the innate difficulty of software engineering: a) The inherent complexity and diversity; b) The difficulty of establishing and stabilizing requirements; c) The changeability or malleability of software; d) The abstraction and intangibility of software products; e) The requirement of varying problem domain knowledge; f) The non-deterministic and poly-solvability in design; g) The polyglotics and polymorphism in implementation; and h) The dependability of interactions between software, hardware, and human beings.

In addition to the fundamental cognitive characteristics, a set of domain specific principles of software engineering has been identified, which determine the difficulty of software development and a broad knowledge requirement for software engineers, as follows: a) Problem domain is infinite that includes all application areas of all existing science and engineering disciplines; b) Software engineering is design intensive opposed to repetitive and mass production; c) Application development is a one-off activity; d) Development processes are relatively stable and repetitive; e) A soft-
Software implementation is only one of all possible solutions for a real-world problem on the basis of constraints and tradeoffs; and f) Software engineering needs new forms of denotational mathematics that are different from current analytic ones (Wang, 2007a, 2008a).

The most significant and unique characteristic of software engineering relies on the fact that its problem domain is infinite, because it encompasses almost all domains in the real world, from scientific problems and real-time control to word processing and games. It is significantly larger than any specific and limited problem domain of other engineering disciplines. This stems from the notion of a computer as a universal intelligent machine, and is a feature fundamentally dominating the complexity in engineering design and implementation of greatly varying software systems.

**Software Engineering Psychology**

Software engineering psychology is a transdisciplinary branch of software engineering and cognitive psychology. It is perceived that the nature of software and software engineering is in many ways closer to cognitive psychology than engineering and technology, because software is intangible and complicated abstract artifacts created by human brains and the best software often takes advantages of human creativity (Weinberg, 1971; Wang, 2007a).

A set of personality traits in software engineering psychology is identified for software engineers (Wang, 2007a), such as:

- Multilayered abstract-level thinking
- Imagination of static descriptions in terms of dynamic behaviors
- Organization capability
- Cooperative and team working attitude
- Long-term focus of attentions
- Preciseness
- Reliability
- Expressive capability in expressions and communications

Software engineering psychology identifies two types of programmers in a psychological view: the realistic and idealistic ones. The former may be suitable for coding, testing, and quality control; while the latter are good at solution seeking, Graphic User Interface (GUI) design, and carrying out tasks as system analysts.

It is interesting to contrast and analyze the differences between professionals and amateurs in software engineering. Professional software engineers are persons with professional cognitive models and knowledge on software engineering. They are trained with: a) Fundamental knowledge that governs software and software engineering practices; b) Basic principles and laws of software engineering; c) Proven algorithms; d) Problem domain knowledge; e) Problem solving experience; f) Program developing tools/environments; g) Solid programming skills in multiple programming languages; and h) A global and insightful view on system development, including its required functionalities as well as exception handling and fault-tolerance strategies. However, amateurish programmers are persons who know only one or a couple of programming languages but lack formal training characterized as follows: a) Ad hoc structure of programming knowledge; b) Limited programming experience and skills; c) Eager to try what is directly required before a system architecture is designed; and d) Tend to focus on details without a global and systematic view. Therefore, professional trainings on the foundations of software engineering are a key to improve the qualification of software engineers.

**Cognitive Skill and Experience Transformation in Software Engineering**

In discussing “what makes a good software engineer” in a panel, Marcia Finfer (1989) believed: “the answer, in my opinion, is simply the combination of both innate skill and significant experience in building real systems against a set of functional and performance requirements.
and a given budget and schedule.” This shows that professional experience is a primary factor of software engineers, where an experience of problem complexity beyond 5,000LOC is a necessary benchmark (Wang, 2007a). Also, the possession of fundamental principles and laws of software engineering is essential towards excellent software engineers.

According to the cognitive informatics model (Wang, 2007c), although knowledge can be acquired indirectly in learning, experiences and skills must be obtained directly by empirical actions. The acquisition of professional skills may be described by a cognitive process. For example, in a complex building, if a newcomer is guided through once, he or she may still have difficulty to manage to remember the ways in the beginning. Because an abstract model of the building, known as the cognitive map, has yet to be built in his/her Long-Term Memory (LTM) and which takes time according to the 24-hour law of memory (Wang and Wang, 2006). The acquisition of skills for driving is another example that explains skill acquisition according to cognitive informatics principles.

It is curious to seek what made skill and experience transfer hard in software engineering, because it is observed that programming skills and software engineering experiences cannot be transferred directly from person to person without hands on practice. The means of experience repository in software engineering can be categorized into the following types: a) Best practices; b) Know-how; c) Lessons learnt; d) Failure reports; and e) New technology trial reports. All the above items of software engineering experience seem to be hard to gain indirectly by reading because the following reasons: (a) The human brain has no direct knowledge transfer mechanism. Each brain is determined by the differences of unique individual’s physiology, cognitive style, personality, and environment. Therefore, any experience or practical knowledge as new information has to be personally acquired and represented as a cognitive Object-Attribute-Relation (OAR) model in the brain (Wang, 2007c), which is linked to existing knowledge in the hierarchical neural clusters of long-term memory; (b) The only way for hand-on skill transformation is learn by doing, or trial and error; and (c) People, usually, have to experience mistakes in order to learn and remember a specific experience. In addition, it is recognized that each brain is sufficiently unique (Wang, 2007a) because of individual physiological differences, cognitive style differences, personality differences, and learning environment differences.

Therefore, although computers as external or extended memory and information processing systems for the brain provide a new possibility for people to learn things faster than ever, the internal representation of abstract knowledge or active behaviors such as skills and experiences must still rely on wired inter-connections among neural clusters in the brain.

**Theoretical Foundations of Quality Assurance in Software Engineering**

On the basis of various fault-tolerant measures (Wang, 2008e), the following statistical properties of human errors may be observed.

**Theorem 5.** The statistical properties of human errors are as follows:

a. **Oddness:** Although individuals make different errors in performing tasks, the chance of making a single error for a given task is most of the cases than that of multiple errors.

b. **Independence:** Different individuals have different error patterns in performing the same task.

c. **Randomness:** Different individuals do not often make the same error at the same times in performing tasks.

Properties (a) through (c) reveal the random nature of human errors on object, action, space, and time in performing tasks in a group.

**Corollary 2.** The random nature of human errors in task performing in a group is determined
by the statistical properties that the occurrences of same errors by different individuals are most likely at different times.

The findings as stated in Theorem 5 and Corollary 2 form a theoretical foundation for fault-tolerance and quality assurance in software engineering. The model indicates that human errors may be prevented from happening or be corrected after their presence in a coordinative group context by means of peer reviews.

**Theorem 6.** The n-fold error reduction structure states that the error rate of a work product can be reduced up to n folds from the average error rate of individuals \( r_e \) in a coordinative group via n-nary peer reviews based on the random nature of error distributions and independent nature of error patterns of individuals, i.e.:

\[
R_e = \prod_{k=1}^{n} r_e(k) \quad (9)
\]

**Example 1.** A software engineering project is developing by a group of four programmers. Given the individual error rates of the four group members as: \( r_e(1) = 10\% \), \( r_e(2) = 8\% \), \( r_e(3) = 20\% \), and \( r_e(4) = 5\% \), estimate the error rates of final software by adopting the following quality assurance techniques: (a) Pairwise reviews between Programmers 1 vs. 2 and Programmers 3 vs. 4; and (b) 4-nary reviews between all group members.

a. The pairwise reviews between Programmers 1-2 and Programmers 3-4 will result in the following error rates \( R_{e1} \) and \( R_{e2} \):

\[
R_{e1} = \prod_{k=1}^{2} r_e(k) = 10\% \cdot 8\% = 0.8\%
\]

\[
R_{e2} = \prod_{k=3}^{4} r_e(k) = 20\% \cdot 5\% = 1.0\%
\]

b. The 4-nary reviews between Programmers 1 through 4 will yield the following error rate \( R_{e3} \):

\[
R_{e3} = \prod_{k=1}^{4} r_e(k) = 10\% \cdot 8\% \cdot 20\% \cdot 5\% = 0.008\%
\]

Theorem 6 and Example 1 explain why multiple peer reviews may greatly reduce the probability of errors in program development and software engineering. Theorem 6 is also applicable in the academic community, where peer-reviewed results may virtually prevent any mistake in a final article before its publication.

In software engineering quality assurance, a four-level quality assurance system is needed for certain critical software functions and projects as shown in Table 4.

**Example 2.** For a given program reviewed according to the four-level quality assurance system as shown in Table 4, assuming \( r_e(10) = 10\% \), \( r_e(2) = 5\% \), \( r_e(3) = 2\% \), and \( r_e(4) = 10\% \), estimate the quality of the final result of this program.

According to Eq. 9, the 4-nary quality assurance system may yield an expected error rate \( R_{e4} \):

\[
R_{e4} = \prod_{k=1}^{4} r_e(k) = 10\% \cdot 5\% \cdot 2\% \cdot 10\% = 0.001\%
\]

The results indicate that the error rate of the above system has been significantly reduced from initial 100bugs/kLOC to 1bug/kLOC. This demonstrates that the hierarchical organization form for software system reviews can greatly increase the quality of software development and significantly decrease the requirement for individual capability and error rates in software engineering.
Table 4. The four-level quality assurance system of software engineering

<table>
<thead>
<tr>
<th>Level</th>
<th>Checker</th>
<th>Means</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Programmer</td>
<td>Self checking, module-level testing</td>
</tr>
<tr>
<td>2</td>
<td>Senior member</td>
<td>Peer review, module-level testing</td>
</tr>
<tr>
<td>3</td>
<td>Tester/quality engineer</td>
<td>System-level testing, audit, review, quality evaluation</td>
</tr>
<tr>
<td>4</td>
<td>Manager</td>
<td>Quality review, deliver evaluation, customer survey</td>
</tr>
</tbody>
</table>

Table 5. Definitions of BCS’s and equivalent cognitive weights ($W_i$)

<table>
<thead>
<tr>
<th>Category</th>
<th>BCS</th>
<th>Structure</th>
<th>$W_i$</th>
<th>RTPA notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>Sequence (SEQ)</td>
<td><img src="image" alt="Sequence diagram" /></td>
<td>1</td>
<td>P → Q</td>
</tr>
<tr>
<td>Branch</td>
<td>If-then-else (ITE)</td>
<td><img src="image" alt="Branch diagram" /></td>
<td>3</td>
<td>($\exp BL = T$) → P \lor ($\neg$) → Q</td>
</tr>
<tr>
<td></td>
<td>Case (CASE)</td>
<td><img src="image" alt="Case diagram" /></td>
<td>4</td>
<td>$\exp BT = 0 \rightarrow P_0 \lor 1 \rightarrow P_1 \lor \ldots \lor n-1 \rightarrow P_{n-1} \lor \text{else} \rightarrow \emptyset$</td>
</tr>
<tr>
<td>Iteration</td>
<td>For-do (R_i)</td>
<td><img src="image" alt="Iteration diagram" /></td>
<td>7</td>
<td>( \frac{1}{i} R(P(i)) )</td>
</tr>
<tr>
<td></td>
<td>Repeat-until (R_i)</td>
<td><img src="image" alt="Iteration diagram" /></td>
<td>7</td>
<td>P \→ ( \frac{1}{exp BL = F} R(P) )</td>
</tr>
<tr>
<td></td>
<td>While-do (R_o)</td>
<td><img src="image" alt="Iteration diagram" /></td>
<td>8</td>
<td>( \frac{F}{exp BL = T} R(P) )</td>
</tr>
<tr>
<td>Embedment</td>
<td>Function call (FC)</td>
<td><img src="image" alt="Embedment diagram" /></td>
<td>7</td>
<td>P ⇔ F</td>
</tr>
<tr>
<td></td>
<td>Recursion (REC)</td>
<td><img src="image" alt="Embedment diagram" /></td>
<td>11</td>
<td>P ⊕ P</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Parallel (PAR)</td>
<td><img src="image" alt="Concurrency diagram" /></td>
<td>15</td>
<td>P</td>
</tr>
<tr>
<td></td>
<td>Interrupt (INT)</td>
<td><img src="image" alt="Concurrency diagram" /></td>
<td>22</td>
<td>P \parallel (@e S.P Q . Q) \parallel @)</td>
</tr>
</tbody>
</table>
The Cognitive Complexity of Software

One of the central problems in software engineering is the inherited complexity. Software complexity may be classified into computational (time, space), symbolic (LOC), structural (control flow, cycloomatic), functional (function points, cognitive complexity), and system complexity (McCabe, 1976; Halstead, 1977; Albrecht and Gaffney, 1983; Lewis and Papadimitriou, 1998; Hartmanis, 1994; Wang, 2009c).

In cognitive informatics, it is found that the functional complexity of software can be measured by its cognitive complexity, which is determined by the internal control flows known as the Basic Control Structures (BCS’s) and its input and output (2007a).

**Definition 15.** Basic Control Structures (BCS’s) are a set of essential flow control mechanisms that are used for building logical architectures of software.

**Definition 16.** The cognitive weight of software is the extent of difficulty or relative time and effort for comprehending a given piece of software modeled by a set of BCS’s.

There are only 10 BCS’s in software structures as shown in Table 5. The cognitive weight of each of the BCS’s may be quantitatively measured and calibrated. In Table 5, the relative cognitive weights (W) for determining a BCS’ functionality and complexity are calibrated based on psychological experiments and empirical studies in cognitive informatics and software engineering (Wang, 2007a).

Substantial findings in the study on the cognitive complexity properties of software have been that: a) the cognitive complexity of software in design and comprehension are dependent on three factors – internal processing structures as well as numbers of inputs and outputs; b) The cognitive complexity measure is more robust than the symbolic size measures of software (such as LOC), and independent from languages/implementations; c) Cognitive complexity provides a foundation for cross-platform analysis of complexities and sizes of both software specifications and implementations for either design or comprehension purposes in software engineering; d) Although there are similar sized software systems in term of symbolic sizes in LOC, their functional sizes in cognitive complexity would be dramatically different; and e) When the symbolic size of software systems grows above 100LOC, its cognitive complexity could be increased exponentially due to intricate interconnections of highly complicated BCS’s (Wang and Shao, 2003).

**CONCLUSION**

Beveridge (1957) once questioned that “Elaborate apparatus plays an important part in the science of today, but I sometimes wonder if we are not inclined to forget that the most important instrument in research must always be the mind of man.” Because software engineering involves intensive human creative work, the studies on the nature of software and the cognitive informatics foundations of software engineering addresses a set of central and essential problems for software engineering and software science.

This paper has explored the cognitive characteristics of software and software engineering practice. As a result, a synergy between software engineering, computer science, and cognitive informatics is established. The nature of software has been modeled by a set of properties of informatics, mathematics, and intelligent behaviors. The cognitive informatics foundations of software engineering have been developed by a set of cognitive constraints and principles. The psychological requirements for software engineers have been identified, such as abstract-level thinking, imagination of dynamic behaviors with static descriptions, organization capability, cooperative attitude in team work, long-time focus of attentions, preciseness, reliability, and expressive capability in communications. A set of key findings in this work on the
cognitive informatics foundations of software engineering has been presented, which cover the cognitive constraints of software engineering, the conservative productivity of software development, cognitive complexity of software, and the cognitive informatics theory of software quality assurance.

ACKNOWLEDGMENT

This work is partially sponsored by the Natural Sciences and Engineering Research Council of Canada (NSERC). The authors would like to thank the anonymous reviewers for their valuable suggestions and comments on this work.

REFERENCES


Arnowitz, J., M. Arent, and N. Berger (2006), Effective Prototyping for Software Makers, Morgan Kaufmann.

Bauer, F.L (1972), Software Engineering, Information Processing, 71.


Wang, Y. (2008j), Deductive Semantics of RTPA,
International Journal of Cognitive Informatics and Natural Intelligence, 2(2), April, pp. 95-121.


Yingxu Wang is professor of cognitive informatics and software engineering, director of International Center for Cognitive Informatics (ICfCI), and director of Theoretical and Empirical Software Engineering Research Center (TESERC) in Dept. of ECE at University of Calgary, Canada. He received a PhD in software engineering from the Notting¬ham Trent University, UK, in 1997, and a BSc in electrical engineering from Shanghai Tiedao University in 1983. He is/was a visiting professor in the Computing Laboratory at Oxford University in 1995, Dept. of Computer Science at Stanford University in 2008, and the Berkeley Initiative in Soft Computing (BISC) Lab at University of California, Berkeley in 2008, respectively. He has been a full professor since 1994. He is a Fellow of WIF, a P.Eng of Canada, a Senior Member of IEEE and ACM, and a member of ISO/IEC JTC1 and the Canadian Advisory Committee (CAC) for ISO. He is the founder and steering committee chair of the annual IEEE International Conference on Cognitive Informatics (ICCI). He is founding editor-in-chief of International Journal of Cognitive Informatics and Natural Intelligence (IJCI), founding editor-in-chief of International Journal of Software Science and Computational Intelligence (IJSSCI), associate editor of IEEE TSMC(A), and editor-in-chief of CRC Book Series in Software Engineering. He has published over 300 peer reviewed journal and conference papers and 12 books in cognitive informatics, software engineering, and computational intelligence. He is the recipient of dozens of research achievement, best paper, and teaching awards in the last 30 years.

Shushma Patel is a professor in information systems at London South Bank University. Her background is in medical sciences and she holds a PhD from the Faculty of Medicine, University of London. She heads the Health Informatics research group within the Centre for Information Management & E-Business. She is a Fellow of the British Computer Society and a Chartered IT Professional (CITP). Her research interests are varied, although there is a strong emphasis on cognitive informatics, knowledge management, decision-making, organizational behaviour and related communication issues in both commercial and educational settings. She has published extensively in journals, book chapters and conference proceedings. In addition she has edited special issues of journals and conference proceedings. She has chaired and served on numerous international conferences and programme committees including IEEE ICCI 2003. As a subject specialist reviewer in computing, she has undertaken subject reviews for the UK Quality Assurance Agency in Higher Education. She has worked on a number of research projects funded by the European Union, Department of Trade and Industry (DTI) and industry.

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.